217921

Output-Sensitive Hidden Surface Elimination for Rectangles

Mikhail J. Atallah
Michael T. Goodrich

September, 1988

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.42

NASA Cooperative Agreement Number NCC 2-387

RIACS

Research Institute for Advanced Computer Science

Output-Sensitive Hidden Surface Elimination for Rectangles

Mikhail J. Atallah[1]

Dept. of Computer Science, Purdue University, West Lafayette, IN 47907

Michael T. Goodrich[2]

Dept. of Computer Science, The Johns Hopkins University, Baltimore, MD 21218

Abstract

We present an algorithm for the well-known hidden-surface elimination problem for rectangles, which is also known as the window rendering problem. The time complexity of our algorithm is sensitive to the size of the output. Specifically, it runs in time that is $O(n^{1.5} + k)$, where k is the size of the output (which can be as large as $\Theta(n^2)$). For values of k in the range between $n^{1.5}/\log n$ and n^2, our algorithm is asymptotically faster than previous ones.

1 Introduction

The hidden-surface elimination problem is well known in computer graphics and computational geometry [6,12,13,15,16,19,20,21,22]: one is given a set of simple, non-intersecting planar polygons in 3-dimensional space, and a projection plane π, and wishes to determine which portions of the polygons are visible when viewed from infinity along a direction normal to π, assuming all the polygons are opaque. An important special case of this problem occurs when the polygons are all *isothetic* rectangles, i.e., the rectangles are all parallel to the xy-plane and have sides that are parallel to either the x- or y-axis. This version of the hidden-surface elimination problem is also known as the *window rendering* problem [4], since it is the problem that must be solved to render the windows that might need to be displayed on the screen of a work-station. (See Figure 1.) Another situation where one often wishes to render such a collection of rectangles is in

[1]This author's research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T. Part of this research was carried out while this author was visiting the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California.

[2]This author's research was supported by the National Science Foundation under Grant CCR-8810568.

Figure 1: (a) isothetic rectangles; (b) their visible portion.

drafting software, where any time a rectangle R_1 is created, by the draftsman, before rectangle R_2 is created, then R_1 is "behind" R_2, unless the draftsman explicitly changes this ordering (e.g., by executing a "move to front" command on R_1 or, equivalently, a "send to back" command on R_2).

Using the terminology of [22], we are interested in the *object space* version of this problem. That is, we want a method that produces a device-independent, mathematically-based representation of the visible surfaces. One reason for our interest in an object space solution is that such a solution is not dependent on a certain method for rendering polygons nor on the number of pixels on a display screen (which seems to grow with each passing year). In addition, an object space solution gives us a representation that is easily scaled and rotated.

We briefly review some of the efficient algorithms for the window rendering problem. Since this problem is a special case of hidden-surface elimination [13], any algorithm for the general case can also be used for this problem. In [13] McKenna shows how to solve the general hidden-surface elimination problem in $O(n^2)$ time, generalizing an algorithm by Dévai [6] for the easier hidden-line elimination problem that also runs in $O(n^2)$ time. (In the hidden-line elimination problem one is only interested in computing the portions of the polygonal boundaries that are visible.) Both of these algorithms are worst-case optimal, because there are problem instances that have $\Theta(n^2)$ output size (e.g., a collection of rectangles that form a cross hatched pattern, as in Figure 2a.)

Unfortunately, these algorithms always take $O(n^2)$ time, even if the size of the output is very small (e.g., $O(1)$). There are algorithms that run faster than $O(n^2)$ for certain

Figure 2: (a) Quadratic output size; (b) Small output size with quadratic I.

problem instances, however. We review these next.

In [15] Nurmi gives an algorithm for general hidden-line elimination that runs in $O((n+I)\log n)$ time and $O((n+I)\log n)$ space, where I is the number of pairs of line segments whose projections on π intersect (I is $O(n^2)$). Schmitt [19] is able to achieve this same time bound for hidden-surface elimination using only $O(n+I)$ space. If I is $o(n^2/\log n)$, then these algorithms clearly run faster than $O(n^2)$ time. Their worst-case performance is, however, a suboptimal $O(n^2 \log n)$ time (if I is $\Theta(n^2)$).

In [12] Güting and Ottmann address the window rendering problem (they are probably the first to study this important special case of hidden-surface elimination), giving an algorithm that runs in $O(n \log^2 n + I)$ time. In [9] Goodrich shows how to solve general hidden-line elimination, and a version of hidden-surface elimination that includes the window rendering problem as a special case, in $O(n \log n + I + P)$ time, where P is the number of pairs of polygons whose projections on π intersect (P is $O(n^2)$). Both of these algorithms are optimal in the worst case and also take advantage of problem instances that are "simpler" than in the worst case, but they are not truly output-sensitive. Indeed, there are problem instances where these two algorithms run in $O(n^2)$ time even though the output size is very small (e.g., a large rectangle that covers up a collection of cross hatched rectangles, as in Figure 2b.)

Recently, Bern [4] and Preparata, Vitter, and Yvinec [18] have shown that one can solve the window rendering problem in $O(n \log n \log\log n + k \log n)$ time and $O(n \log^2 n + k \log n)$ time, respectively, where k is the actual size of the output (recall that k is at

worst $\Theta(n^2)$). Thus, they have shown that one can solve the window rendering problem in an output-sensitive manner. Their algorithms are not worst-case optimal, however.

In this paper we give an algorithm for the window rendering problem that is both worst-case optimal and output-sensitive. Specifically, our algorithm runs in $O(n^{1.5} + k)$ time, where k is the actual size of the output. Thus, our algorithm is faster than those of Bern [4] and Preparata, Vitter, and Yvinec [18] for k in the range between $n^{1.5}/\log n$ and n^2. Our algorithm is based on a problem-division approach to hidden-surface elimination. In this approach one typically divides the problem—call it A—into two dissimilar subproblems B and C, solves B and C independently (usually by completely different techniques), and then "marries" the solutions to B and C to give a solution to A. Applying this approach to the window rendering problem can lead to an algorithm that runs in $O(n^{1.5}\log n + k\log n)$ time, although the details are somewhat non-trivial. This, of course, is worse than previous solutions for all values of k. One of the ways we avoid these logarithmic multiplicative factors is by modifying the approach so that we divide A into B and C, and solve B, just as before, but then solve C *while* marrying the solutions to B and C. Other ways we avoid these factors are based on fundamental paradigms from computational geometry, including *batched dynamic searching* [8], *space-sweeping* [17], and *fractional cascading* [5].

In the next section we give a high-level description of our algorithm, and in the subsequent sections (3–5) show how to implement each of its constituent steps. We conclude in Section 6.

2 An Overview of the Window Rendering Algorithm

Suppose we are given a collection S of n non-intersecting isothetic rectangles in $\Re^3$, i.e., a collection of rectangles parallel to the xy-plane such that all edges are parallel to either the x- or y-axis. The problem is to compute all the portions of each rectangle that are visible from $z = \infty$ with light rays that are parallel to the z-axis (i.e., the projection plane is the xy-plane).

More specifically, each rectangle R is given by a triple $((x_1, y_1), (x_2, y_2), z)$, where (x_1, y_1) is the lower-left corner of R, (x_2, y_2) is the upper-right corner of R, and z is the z-coordinate of the plane to which R belongs. For the remainder of this paper we assume that the relationships "to the left of" and "to the right of" are with respect

to x-coordinates, that the relationships "above" and "below" are with respect to y-coordinates, and that the relationships "in front of" and "behind" are with respect to z-coordinates.

There are many ways that one can specify what constitutes a solution to the hidden-surface elimination problem [12,13,16,20,21,22]. Let G be the planar subdivision determined by a solution to the hidden-line elimination problem. Typically, a solution to the hidden-surface elimination problem is given by G, augmented so that each polygonal face of G stores the name of the rectangle of S that is visible in that face. Our exposition will gain in simplicity if our output specification, which we denote by $Vis(S)$, generalizes this so that each face is itself a rectangle (our $Vis(G)$ is obtained from G by adding to it a small number of extra edges, as explained below).

We begin our definition of $Vis(S)$ by examining the subdivision G a little more closely. For each vertex v of G either v corresponds to a (visible) *corner point* of a rectangle in S or v corresponds to an intersection of two visible edges (where one of them becomes occluded by the other, i.e., an intersection of the form $\top$, $\perp$, $\vdash$, or $\dashv$). We call such intersections *dead ends*, and classify them into two types: *vertical dead ends*, where the terminating segment is vertical (i.e., $\top$ or $\perp$), and *horizontal dead ends*, where the terminating segment is horizontal (i.e., $\vdash$ or $\dashv$). In Figure 1b, points e and f are corners, a is a $\top$, b is a $\perp$, c is a $\vdash$, and d is a $\dashv$. In that same figure, points a, b, c and d are dead ends: a and b are vertical dead ends, while c and d are horizontal dead ends. For each corner point v in G, extend a horizontal ray from v in the direction that points away from the rectangle to which v belongs. Thus, in Figure 1b, the ray emanating from e goes leftward, whereas that from f goes rightward. The point on the first (vertical) edge of G that is intersected by this ray is known as the *horizontal shadow* of v (if no such intersection with the ray occurs, i.e., the ray continues to infinity, then we consider the point at infinity to be the horizontal shadow of v). Call the new subdivision created from G by drawing an edge from each corner point to its horizontal shadow the *rectangular decomposition* of G, and let G' denote this subdivision. Obviously each face of G' is rectangular rather than polygonal. Figure 3 shows the G' that results from the G of Figure 1b. In that figure, the horizontal shadow of e is g, that of f is at $(+\infty, y(f))$. Our characterization, $Vis(S)$, of a solution to the hidden-surface elimination problem for S consists of the subdivision G' augmented so that each rectangular face of G' stores the name of the rectangle of S that is visible in that face.

Figure 3: The subdivision G'. The edges joining corners to their shadows are shown dotted.

By defining $Vis(S)$ in this way we get a characterization that consists entirely of rectangular faces, yet is at most twice the size of G. For many applications, our specification should lead to simpler rendering algorithms, e.g., by simplifying scan-line conversion.

For convenience, we assume throughout the paper that the planar graph $Vis(S)$ lies in the xy-plane, so that any rectangular face of $Vis(S)$ is also in the xy-plane. Of course, each such rectangular face knows which rectangle of S is visible in it, and the z-coordinate of that rectangle (throughout the paper, each rectangular face of a $Vis(S)$ is always assumed to have, attached to it, which rectangle of S is visible in it).

There are a number of ways one can represent an embedded planar graph, such as $Vis(S)$. Three such representations are the "winged edge" structure of Baumgart [2], the "quad edge" structure of Guibas and Stolfi [11], and the "doubly-connected edge list" structure of Muller and Preparata [14,17]. Our algorithm does not depend on which representation one chooses, so long as the representation allows one to determine each of the following in time proportional to its size:

1. all edges and faces adjacent to a given vertex v, as well as their orientation with respect to v,
2. all vertices and faces adjacent to a given edge e, as well as their orientation with respect to e, and
3. all vertices and edges that lie on the boundary of a given face f, in the order they occur around f.

Each of the mentioned representations provides this.

Given an isothetic rectangle R in $\Re^3$ we let $z(R)$ denote the z-coordinate of the plane to which R belongs. Similarly, for any point p in $\Re^3$, we use $x(p)$, $y(p)$, and $z(p)$ to denote the x-, y-, and z-coordinate of p, respectively. Our terminology implicitly assumes that the observer looking at the scene from $z = \infty$ has his body parallel to the y-axis, with both arms extended so they are parallel to the x-axis (the reader probably inferred this from the way we drew Figure 1b). Hence a *vertical* segment is parallel to the y-axis, whereas a *horizontal* segment is parallel to the x-axis. Similarly, we say that a plane is *vertical* (resp., *horizontal*) if it is parallel to the yz-plane (resp., xz-plane). In addition, we assume that the x-, y-, and z-coordinates of all rectangle endpoints are integers in the range $[1, 2n]$. If this is not the case, then we apply a pre-processing step that, in turn for each of the three coordinates, sorts its values in increasing order and replaces each old value by its rank in the sorted list. This takes $O(n \log n)$ time [1]. For the sake of simplicity, we assume that the x-coordinates of the rectangles' endpoints are distinct, and similarly for y-coordinates and for z-coordinates. Modifying our algorithm for the general case is straightforward, and is left to the interested reader.

The algorithm we outline below constructs $Vis(S)$.

The Hidden Surface Elimination Algorithm (High-Level Description):

Step 1. Problem division. In this step we divide the endpoints of the rectangles of S by vertical planes into r groups, each of size $\lceil 4n/r \rceil$ (with the possible exception of the last group, which may be smaller). Note that this also divides $\Re^3$ into r regions, each delimited by two vertical planes (except for the first and last such regions, which are delimited by only one such plane). We call these regions *slabs*, and let $(\Pi_1, \Pi_2, ..., \Pi_r)$ denote the collection of slabs listed from left to right. For each Π_i we construct $Endpoint_i$ and $Span_i$, where $Endpoint_i$ denotes the set of all rectangles that have at least one endpoint in Π_i, and $Span_i$ denotes the set of all rectangles that span Π_i (i.e., all rectangles that intersect Π_i but do not have an endpoint in it). (See Figure 4.) Let S_i be obtained from $Span_i$ by replacing every rectangle R in $Span_i$ by $R \cap \Pi_i$. Similarly, let E_i be obtained from $Endpoint_i$ by replacing every rectangle R in $Endpoint_i$ by $R \cap \Pi_i$. This step can easily be performed in $O(rn)$ time.

Step 2. Computing $Vis(E_i)$. In this step we solve the hidden-surface elimination problem for each E_i, ignoring all rectangles not in E_i. This can be done in $O((n/r)^2)$ time for each E_i using the algorithm by McKenna [13]. In addition, for each E_i we

Figure 4: R is in $Endpoint_1$, $Span_2$, $Span_3$, and $Endpoint_4$.

perform some preprocessing to help us perform the space-sweeping method of Step 3 (given below). The total time complexity of this step is $O(rn + n^2/r)$, and its details can be found in Section 3.

Step 3. Determining visible corners and vertical dead ends. In this step we determine all corners and vertical dead ends that belong to $Vis(S)$, and for each such point we determine the rectangles of S that are visible in its vicinity (i.e., that are visible in the faces of $Vis(S)$ adjacent to it). In addition, for each corner point p we find the horizontal shadow of p in $Vis(S_i \cup E_i)$, where $p \in \Pi_i$ and its horizontal shadow is now constrained to be in Π_i (so that the horizontal rays by which we defined shadows are stopped by the boundary of Π_i, instead of being allowed to proceed to infinity). We call this the *restricted horizontal shadow* of p. The main idea of our method for performing this step is to perform a space-sweeping procedure that simultaneously sweeps through all the slabs $\Pi_1,...,\Pi_r$ to determine all the visible corners and vertical dead ends. This step requires $O(n \log^2 n + rn + n^2/r + k')$ time, where k' is the total number of (visible) points discovered in the sweep (note that $k' \leq k$). Its details are given in Section 4.

Step 4. Determining visible horizontal dead ends. In this step we repeat Steps 1–3, except that the roles of the x-axis and y-axis are interchanged, that is, we divide by horizontal planes and sweep horizontally. We do not perform the extra work, as done in Step 3, to find visible corners and their shadows, however. That is, this step simply discovers all visible horizontal dead ends, and, for each one, all the the rectangles of S that are visible in its vicinity (i.e., that are visible in the faces of $Vis(S)$ adjacent to it).

Step 5. Constructing $Vis(S)$. In this step we combine the information computed in Steps 3 and 4 to construct a representation of $Vis(S)$. Since we have already computed all the visible vertices in $Vis(S)$, we begin by constructing the subdivision G that they determine. We do this using two calls to a bucket sorting routine [1], which takes $O(n+k)$ time. To complete the construction of $Vis(S)$, we must augment G with the true horizontal shadows of all visible corner points. (Recall that Step 3 only yields the restricted horizontal shadow of each corner point p, that is, the horizontal shadow of p restricted to the slab to which p belongs.) The main idea of our method for doing this involves the construction of left and right "horizontal exposure" lists for each Π_i, and the application of the fractional cascading technique [5] to these lists. This gives us a data structure that enables us to find each horizontal shadow in $O(\log n + r)$ time, and then finish constructing $Vis(S)$ in a further $O(n)$ time. Performing the entire step requires $O(n \log n + rn + k)$ time. The details are in Section 5.

End of High-Level Description.

Assuming that we can perform each of the above steps correctly in the stated time bounds, this method gives us an algorithm that runs in $O(n \log^2 n + rn + n^2/r + k)$ time, where k is the size of the output. Setting $r = \sqrt{n}$ gives us the time bound of $O(n^{1.5} + k)$ that we claimed in the introduction.

Let us now give the details for each of the above steps. The details of Step 1 should be obvious given the above description, so we begin our discussion with Step 2.

3 Step 2: Computing $Vis(E_i)$, and preparing for Step 3

Recall that in Step 2 we wish to solve the hidden-surface elimination problem for each E_i in $O((n/r)^2)$ time. Since each E_i contains $O(n/r)$ rectangles, this amounts to being able to perform hidden-surface elimination in time that is quadratic in the number of rectangles. As mentioned above, we can do this by calling the algorithm of McKenna [13] as a subroutine. This section, however, does more than just call McKenna's algorithm: it computes information that will be crucial to the efficient implementation of Step 3. For that purpose, we need to briefly review McKenna's method and somewhat modify its output.

When applied to a set S of isothetic rectangles, McKenna's method constructs the arrangement in the xy-plane produced by (i) extending each rectangular edge to infinity

Figure 5: The arrangement resulting from Figure 1.

in each direction, (ii) projecting the lines so obtained on the xy-plane, and (iii) determining the rectangle of S visible in each rectangular face of the arrangement produced by these projected lines. Figure 5 shows the arrangement resulting from the situation depicted in Figure 1 (in boldface are the edges of the arrangement that are projections of edges of rectangles in S).

Suppose we have already applied McKenna's method to E_i, producing $W(E_i)$. First we use the boundary of Π_i to "clip" all the infinite horizontal edges of $W(E_i)$ (i.e., they now stop at this boundary instead of proceeding to infinity). Then we delete from $W(E_i)$ all the segments that are not in $Vis(E_i)$, i.e., we eliminate each edge e that has the same rectangle of E_i visible on both sides of e, unless e joins a corner to the (restricted) horizontal shadow of that corner (recall that segments that extend from corner points to their respective horizontal shadows are part of $Vis(E_i)$). This is easily done by checking whether both of the two faces of $W(E_i)$ that are adjacent to e have the same rectangle of E_i visible in both of them, and whether e joins a corner to its (restricted) horizontal shadow.

We now do some preprocessing that will help us efficiently implement Step 3. In Step 3 we will be performing a space-sweeping procedure in which we sweep a horizontal plane λ in the negative y-direction. At certain *events* during this sweep we will need to update some dynamic data structures associated with the slab Π_i. The preprocessing we do now facilitates our being able to perform these update operations efficiently. Basically, we take advantage of the fact that the set of operations we will be performing on these dynamic data structures are known in advance, i.e., it is a *batched* problem. In gen-

eral, this paradigm of taking advantage of the batched nature of the dynamic problems that arise in geometric problems such as ours is known as *batched dynamic searching*. Applications of this paradigm to other geometric problems are given by Edelsbrunner and Overmars [8], for example.

The details of our preprocessing steps are as follows. Suppose we wish to sweep a horizontal plane λ through $Vis(E_i)$ in the negative y-direction (as will happen for real in Section 4). Such a plane would encounter $t = O(n/r)$ horizontal positions, each of which coincides with (possibly many) horizontal edges of $Vis(E_i)$. Each horizontal position determines a horizontal plane, which corresponds to a "snap shot" of the plane λ at the time it would encounter that position. The collection of all such horizontal planes divides Π_i into $t+1$ regions, which we call *strips*. Number these strips, from top to bottom, 1, 2, 3, and so on. Thus, the strips form a horizontal partitioning of Π_i. Now, for each rectangular face f of $Vis(E_i)$ that intersects (say) the t_1 strips numbered $t_0, t_0+1, \ldots, t_0+t_1-1$, create t_1 copies of f, each copy being associated with one of those t_1 strips. The copy of f associated with strip s gets assigned as its *key* the pair (s,z) where z is the z-component of the rectangle of E_i that is visible in rectangular face f. (Thus all the copies of f have keys with the same second component.) Observe that the sum over all $f \in Vis(E_i)$ of the number of (s,z) pairs associated with f is $O((n/r)^2)$, because each strip can determine at most $O(n/r)$ (s,z) pairs, and there are $O(n/r)$ strips.

Let C_i denote the collection of (s,z) pairs, where each (s,z) pair contains a pointer to the face f in $Vis(E_i)$ associated with that pair. Now, bucket sort C_i using the lexicographical ordering determined by the (s,z) keys for comparisons. This takes $O(n+(n/r)^2)$ time. For each strip s let $Z_{i,s}$ denote the part of this sorted list that has s as its key's first coordinate. For each s compare the list $Z_{i,s}$ with the list $Z_{i,s+1}$, constructing three sorted lists, $Same_{i,s}$, $Delete_{i,s}$, and $Insert_{i,s+1}$, defined as follows. The list $Same_{i,s}$ consists of all the rectangular faces f that have a copy in both $Z_{i,s}$ and $Z_{i,s+1}$ (the key of f in $Same_{i,s}$ is inherited from the copy in $Z_{i,s}$ rather than from that in $Z_{i,s+1}$). The list $Delete_{i,s}$ consists of all the rectangular faces f that have a copy in $Z_{i,s}$ but not in $Z_{i,s+1}$ (the key of f in $Delete_{i,s}$ is the same as its key in $Z_{i,s}$). The list $Insert_{i,s+1}$ consists of all the rectangular faces f that have a copy in $Z_{i,s+1}$ but not in $Z_{i,s}$ (the key of f in $Insert_{i,s+1}$ is the same as its key in $Z_{i,s+1}$).

Note that the keys of the elements of $Same_{i,s}$ all have the same first component

(namely, s), so that the contents of $Same_{i,s}$ are in fact sorted by the second components of their respective keys (that is, by the z-coordinates of the respective rectangles of E_i that are visible in them). Therefore, from now on, we shall ignore the first components of the keys of the elements of $Same_{i,s}$. That is, a key is from now on a z-value rather than a pair (s, z). Similar remarks hold for each $Delete_{i,s}$, and also for each $Insert_{i,s}$.

For each rectangular face f in $Insert_{i,s+1}$, we determine its predecessor in $Same_{i,s}$ and store this in a field $Pred_{i,s+1}(f)$ associated with f.

Once this is completed, we no longer need the $Same_{i,s}$ lists. The $Delete_{i,s}$ and $Insert_{i,s}$ lists, on the other hand, will become very helpful in performing the space-sweeping procedure in Step 3. Specifically, we shall use them to maintain a list (the *current list*) of rectangular faces of $Vis(E_i)$ that are intersected by a horizontal plane λ (the plane we use for sweeping in the negative y direction). That is, to move some such plane λ from the strip s to the strip $s+1$ we need only consult the lists $Delete_{i,s}$ and $Insert_{i,s+1}$ to tell us which rectangular faces to respectively delete and insert from the current list. In addition, by storing, in the field $Pred_{i,s+1}(f)$, the predecessor in $Same_{i,s}$ of each rectangular face $f \in Insert_{i,s+1}$, we enable ourselves to perform the insertion of f into the current list in $O(1)$ time. Section 4 contains the details of how all these things are done.

The computation of $Vis(E_i)$ and of the $Delete_{i,s}$ lists and $Insert_{i,s}$ lists (and their associated $Pred_{i,s}$ fields) takes $O((n/r)^2)$ time for each E_i, and hence the total time complexity of Step 2 is $O(r(n/r)^2) = O(n^2/r)$.

We next show how to combine the information of the previous two steps to implement Step 3.

4 Step 3: Computing visible corners and visible vertical dead ends

In this step, we use the information computed in the previous step to aid in the implementation of a space-sweeping procedure that computes (i) all the corners and vertical dead ends in all the $Vis(S_i \cup E_i)$'s, (ii) for each such point, the rectangular faces of $Vis(S_i \cup E_i)$ that are adjacent to it, and (iii) for each such rectangular face, the rectangle of $E_i \cup S_i$ (and hence of S) that is visible in it.

We implement this step by sweeping space in the negative y-direction with a hori-

zontal plane λ. We will be using a number of data structures to implement this space sweep:

1. For each Π_i we maintain a variable $CurVis_i$ that stores the name of the rectangle of $Span_i$ having highest z-coordinate among all elements of $Span_i$ currently intersected by λ. Note: we never maintain all of $Vis(S_i)$, just $CurVis_i$.

2. For each Π_i we maintain a list D_i that stores all the rectangular faces of $Vis(E_i)$ that are currently intersected by λ, sorted by non-increasing z-coordinates of the rectangles of E_i visible in them (that is, by associating with each face the z-coordinate of the rectangle of E_i that is visible in it). Each D_i is represented using a doubly linked list and has an entry-pointer (or "finger") f_i that points to the last face in D_i whose associated z-value is greater than $z(CurVis_i)$. If there is no such face in D_i, then f_i points to the first element in D_i. We also maintain for each Π_i an array of pointers called $Where_i$, such that for every rectangular face $f \in Vis(E_i)$, $Where_i(f)$ points to the location of f in D_i if $f \in D_i$, and is nil otherwise.

3. We maintain a tree T that contains the set S_λ of rectangles in $\cup_{i=1}^{r} Span_i$ that are currently intersected by λ. (Note that S_λ contains no more than n elements, since it is a subset of S.) The tree T is represented by a *priority segment tree* [3,12], where the leaves of T are associated with the slabs $\Pi_1, \Pi_2, \ldots, \Pi_r$, listed from left to right. For each internal node v of T we associate a slab $\Pi(v)$ that is the union of the slabs associated with the descendents of v in T. Let the i-th leaf be v_i, so that $\Pi(v_i) = \Pi_i$. In addition, for each node v we store a list $Cover(v)$ that stores all the rectangles of S_λ that span $\Pi(v)$ but do not span $\Pi(parent(v))$, sorted by decreasing z-coordinates. Each list $Cover(v)$ is represented by a dynamic tree structure (e.g., a (2,3)-tree [1] or a red-black tree [10,23]) augmented with a pointer to the rectangle in $Cover(v)$ with largest z-coordinate (we call it $Max(v)$). Every node v also stores $Best(v)$, which is the rectangle that has maximum z-coordinate in the set of $Max(w)$'s stored in the nodes on the path from v to the root of T. It is not hard to see that a rectangle R can appear in no more than $2 \log r$ different $Cover(v)$'s, so that the space complexity of T is $O(r + |S_\lambda| \log r) = O(r + n \log r)$.

The following lemma follows immediately from the above definitions.

Lemma 4.1: *Assuming the above data structures are correctly maintained for the current position of λ, then, for any Π_i, $CurVis_i$ is equal to $Best(v_i)$ where v_i is the leaf that corresponds to Π_i in T.*

Proof: An immediate consequence of the definitions. ∎

Let Y be the list of the $2n$ horizontal edges of the rectangles in S, sorted in decreasing order of their y-coordinates. For each edge in Y we store the name of the rectangle that determined that edge. The list Y determines the *events* in the space sweep. The goal of the sweep procedure is to discover all visible corner points and visible vertical dead ends. Initially, $CurVis_i$ is set to the "background" rectangle $-\infty$, all the D_i lists are empty, and all the $Cover(v)$ lists in T are empty.

To implement the space sweep we iteratively examine each edge in Y. Suppose l is the next event in Y. Let R be the rectangle of S to which l belongs. Let a (resp., b) be the left (resp., right) endpoint of l. There are essentially two different kinds of updates we must perform for l: updating the slabs containing a and b (Subsection 4.1 below), and updating the slabs spanned by l (Subsection 4.2 below). But before doing any of these, we begin by updating the tree T so that it already reflects the occurrence of event l. This is done as follows.

If l is the upper edge of R then, just after event l, the sweeping plane λ intersects R (whereas it did not before event l) and therefore we must insert R in all the $Cover(v)$'s to which it belongs and update their respective $Max(v)$'s accordingly. As already stated, there are at most $2\log r$ such nodes $v \in T$ whose respective $Cover(v)$'s are to be updated. Since each such update takes $O(\log n)$ time, the amount of time for all such updates is $O(\log r \log n) = O(\log^2 n)$. On the other hand, if l is the lower edge of R then, just after event l, the sweeping plane λ no longer intersects R (whereas it did before event l) and therefore we must delete R from all the $Cover(v)$'s to which it belongs and update their respective $Max(v)$'s accordingly. Of course, this too takes $O(\log^2 n)$ time, by an argument similar to that for the case of insertions. Finally, we must update the $Best(w)$ values in T, so that they reflect the new $Max(v)$ values. This is easily done by a preorder traversal of T during which we maintain a stack A of all the $Best(v)$ values from the current node to the root. That is, if the traversal is at node w_1, and if the path from w_1 to the root of T is $w_1, w_2, \ldots, w_t$, then the stack A contains $Best(w_1), Best(w_2), \ldots, Best(w_t)$ (with $Best(w_1)$ at the top of the stack). It is trivial to maintain the stack A during the traversal of T, as follows. When we traverse down T to

a new node v, we compare $z(A[top])$ to $z(Max(v))$ and push the rectangle achieving the larger of these two onto $A[top]$. When we traverse up T we pop the top element off A. The traversal for updating the $Best(v)$'s takes $O(r)$ time, since T has $O(r)$ nodes. Thus the time for updating T as a result of event l is $O(\log^2 n + r)$.

Now that T is updated, we can proceed to compute the effect of event l on the various Π_i's.

4.1 Processing the "endpoint" slabs

We first describe the updating of D_h where $a \in \Pi_h$. (The updating of D_j, where $b \in \Pi_j$, is similar.) Since a is in Π_h there were two adjacent strips (say, strips s and $s+1$) in Step 2 whose common boundary is the horizontal plane L containing l. In moving λ from s to $s+1$ past this horizontal plane L, we must delete the rectangular faces of $Vis(E_i)$ that will no longer be intersected by λ and insert the new rectangular faces that will become intersected by λ. Determining these rectangular faces is easy, given the preprocessing done in the previous step (Step 2). Suppose we are in strip s and crossing into strip $s+1$ at L. To determine which faces to delete from D_h we need only consult the list $Delete_{h,s}$: for each f in it, we follow the $Where_h(f)$ pointer which tells us where f occurs in D_h and thus enables us to delete f from D_h in $O(1)$ time. To determine which rectangular faces to insert in D_h, we consult the list $Insert_{h,s+1}$: for each f in that list, the $Pred_{h,s+1}(f)$ pointer tells us which rectangular face f' immediately precedes the location in D_h where f is to be inserted, and following $Where_h(f')$ enables us to complete the insertion of f in D_h in $O(1)$ time. Of course after deleting f from D_h we must update the $Where_h$ array by doing $Where_h(f)$:=nil. Similarly, after inserting f in D_h we must change $Where_h(f)$ from being nil to pointing to where f is in D_h. Updating D_h therefore clearly takes $O(|Delete_{h,s}| + |Insert_{h,s+1}|)$, which is $O(|E_h|) = O(n/r)$ because λ coincides with at most $O(|E_h|)$ edges of $Vis(E_h)$.

In addition, for each rectangular face f in $Delete_{h,s} \cup Insert_{h,s+1}$ we perform the following computations for discovering corners and vertical dead ends on f in $Vis(E_i)$ that are also in $Vis(E_i \cup S_i)$. Let e be the projection of l onto f (in the projection plane); recall that since $f \in Delete_{h,s} \cup Insert_{h,s+1}$, e must contain one of the horizontal boundaries of f. For each endpoint p of e, we check if p is in $Vis(E_i \cup S_i)$, by comparing $z(CurVis_i)$ to the z-coordinate of the rectangle of E_i that is visible in f: if $z(CurVis_i)$ is the larger of the two then p is not in $Vis(E_i \cup S_i)$, otherwise it is. If p is in $Vis(E_i \cup S_i)$—

i.e., p is visible—then we must find out, for each rectangular face f' adjacent to p in $Vis(E_i)$, which rectangle of $E_i \cup S_i$ is visible in f'. This is easily done by comparing the rectangle $CurVis_i$ to the rectangle of E_i that was visible in f' (the one with the higher z-coordinate wins).

In addition, if p is a corner point, then we must determine its restricted horizontal shadow. To do this, we start walking from p along the horizontal ray leading to p's shadow in $Vis(E_i)$: we walk through all the rectangular faces of $Vis(E_i)$ that are cut by this horizontal ray until either (i) we first hit a face whose z-coordinate is more than $z(CurVis_i)$, or (ii) we reach the boundary of Π_i. Either of events (i) or (ii) gives us the horizontal shadow of p in $Vis(S_i \cup E_i)$, i.e., the restricted horizontal shadow of p.

The above computations for processing the event l for slab Π_h require a total, over all f in $Delete_{h,s} \cup Insert_{h,s+1}$, of $O(|E_i|) = O(n/r)$ time (this is because even though $Vis(E_i)$ has $O((n/r)^2)$ faces, the number of faces cut by any particular position of the sweeping plane λ is $O(|E_i|)$).

4.2 Processing the "spanned" slabs

Assume that the processing of the "endpoint" slabs Π_h and Π_j has already been done, as explained in Subsection 4.1. This section deals with processing the "spanned" slabs, i.e., the Π_i's for which $R \in Span_i$. (Recall that R is the rectangle of S to which event l belongs.)

Let U denote the set of Π_i's that are affected by event l and thus will need further processing. Thus U consists of the Π_i's whose respective $CurVis_i$'s will change as a result of event l (either $CurVis_i$ was R and will cease to be R, or it was not R and will become R). Finding U is easy to do: Lemma 4.1 implies that the new value of each $CurVis_i$—call it $NewCurVis_i$—just after event l is readily available in the tree T (recall that T has already been updated to reflect event l). Therefore we can easily compute U as follows. For each Π_i, compare $CurVis_i$ to $NewCurVis_i$, i.e., to the $Best(v_i)$ entry available in T: if they are not the same rectangle then include Π_i in U.

For each $\Pi_i \in U$, we perform the following computation. For convenience, in what follows we let R_1 stand for $CurVis_i$, and we let R_2 stand for $NewCurVis_i$. Thus R_1 is the rectangle that is in $CurVis_i$ just before event l, and R_2 is the rectangle that becomes in $CurVis_i$ just after event l ($R_1 \neq R_2$ by definition of U). Note that R will be one of R_1 or R_2 (Figure 6 depicts the case $R_1 = R$). We obtain from D_i the set D_i'

Figure 6: Illustrating the case $z(R_1) > z(R_2)$.

of rectangular faces of $Vis(E_i)$ whose associated z-values fall between $z(R_1)$ and $z(R_2)$. If $z(R_1) > z(R_2)$ (as in Figure 6), then the rectangular faces in D' are not visible (in $E_i \cup S_i$) before l, but become visible after l. Otherwise, if $z(R_1) < z(R_2)$, then the rectangular faces in D' are visible before l, but are not visible after l. In either case, (a portion of) each of these rectangular faces is part of the output, $Vis(E_i \cup S_i)$. For each such rectangular face f, we determine the intersection of f with λ, the sweep plane (or, equivalently, the projection of l onto f). Let $p \in f$ be an endpoint of this intersection, and let e be the vertical line segment of $Vis(E_i)$ that contains p. For each such point p, we find the rectangles of $E_i \cup S_i$ that are visible in the vicinity of p by comparing the two z-values (in $Vis(E_i)$) of the two rectangular faces adjacent to e, to $z(R_1)$ and $z(R_2)$. Note that p forms a visible vertical dead end in $Vis(E_i \cup S_i)$ (and hence in $Vis(S)$). We complete the computations for Π_i by assigning $CurVis_i := R_2$. The processing of each such $\Pi_i \in U$ clearly requires $O(k')$ time, where $k' = |D'|$ (recall that a portion of each face in D' is visible in $E_i \cup S_i$, hence is part of the output).

4.3 Analyzing Step 3

When the slab-sweeping procedure terminates we will have computed all the corner points, restricted horizontal shadows, and vertical dead ends in $Vis(S_i \cup E_i)$. (We prove this in the next lemma.) From the comments made during the detailed presentation of Step 3, it is easy to see that performing this entire step requires $O(n \log^2 n + nr + n^2/r + k)$ time, where k is the size of $Vis(S)$. In the following lemma we establish the correctness of our method so far.

Lemma 4.2: *The previous steps correctly find all corner points and vertical dead ends in every* $Vis(S_i \cup E_i)$ *and, for each such point* p*, correctly determine the rectangles of* $E_i \cup S_i$ *that are visible its vicinity (i.e., in each rectangular face of* $Vis(E_i \cup S_i)$ *that is adjacent to* p*).*

Proof: ($\Rightarrow$:) Suppose p is a corner point or vertical dead end in $Vis(S_i \cup E_i)$, where Π_i is the slab containing p. We wish to show that p will be discovered in the previous steps of the algorithm. Let us treat each of the possible cases.

Case 1. p is a corner point, the projection of a vertex p' of some $R \in S$. Let e be the horizontal edge of R containing p'. Since p is a vertex in $Vis(E_i \cup S_i)$, p must be a vertex in $Vis(E_i)$. Therefore, Step 2 will have computed which rectangle of S is visible in each face that is adjacent to p in $Vis(E_i)$. Therefore when event e is processed by Step 3, that step must indeed discover that p is visible in $Vis(E_i \cup S_i)$ (this follows from the way Step 3 works). Moreover, for each face f of $Vis(E_i)$ that is adjacent to p, Step 3 correctly determines the rectangle of S visible in f when it compares $CurVis_i$ to the rectangle of $Endpoint_i$ that is visible in f and chooses the one with the larger z-coordinate.

Case 2. p is a visible vertical dead end in $Vis(E_i \cup S_i)$, i.e., it is of the form $\top$ or $\bot$. If p is a vertex in $Vis(E_i)$, then an argument similar to that for Case 1 applies. So suppose p is not a vertex in $Vis(E_i)$, i.e., it occurs on the interior of an edge e of $Vis(E_i)$. Obviously e must be vertical so that a portion of it becomes the vertical part of the $\top$ or $\bot$ in $Vis(E_i \cup S_i)$, the horizontal part of the $\top$ or $\bot$ being contributed by the edge of a rectangle in $Span_i$. We continue the discussion assuming that p is, in $Vis(E_i \cup S_i)$, of the form $\bot$ (the argument for when p is of the form $\top$ is similar). As already stated, the $\bot$ that p forms is the intersection of a portion of the (vertical) segment e with a horizontal line segment l that is the projection of an edge l' of a rectangle, call it R_{big}, that spans Π_i. (Note that R_{big} becomes the new $CurVis_i$ just after event l' is processed.) Let f' and f'' be the two rectangular faces of $Vis(E_i)$ that are adjacent to e just above p (i.e., just before event l'). Let R' and R'' be the two rectangles of E_i that are visible in $Vis(E_i)$ in (respectively) faces f' and f''. Then R' and R'' must both have lower z-coordinates than that of R_{big} (because p is a $\bot$ in $Vis(E_i \cup S_i)$). Moreover, at least one of R' and R'' (possibly both) must be visible around p in $Vis(E_i \cup S_i)$, i.e., have a z-coordinate larger than that of the $CurVis_i$ just before event l' (otherwise p could not be a $\bot$). Therefore the search performed by Step 3 for event l' will discover at least

one of f' and f''; hence, discover the intersection of e with l at p and all the rectangles visible in the faces around p.

($\Leftarrow$:) Let p be any point determined to be "visible" by Step 3. We wish to show that this action of Step 3 is correct, i.e., that p is indeed visible in $Vis(S_i \cup E_i)$, where Π_i is the slab containing p. Obviously p must be in $Vis(E_i)$ as well, but not necessarily as a vertex (perhaps as a point along some edge). Any rectangle R that can possibly obstruct p is either in E_i or in $Span_i$. Thus, since we compare p to the rectangles in E_i in Step 2, and, in Step 3, to the rectangle with largest z-coordinate whose projection contains p, p is indeed a visible point. Moreover, since p must correspond to an event in one of the plane sweeps of Steps 2 and 3, by arguments similar to those for the ($\Rightarrow$) part of the proof, we do discover all the rectangles of S that are visible in the faces adjacent to p. This completes the proof. ∎

In Step 4 we repeat the above three steps, except that the roles of the x-axis and y-axis are reversed, and we do not bother performing the extra steps to determine corner points and their respective shadows (this was already done in Step 3). Thus, we find all the visible horizontal dead ends. This gives us all the visible vertices of $Vis(S)$, except those that are horizontal shadows. In the next section we show how to extend the restricted horizontal shadows (in $Vis(S_i \cup E_i)$'s) into true horizontal shadows (in $Vis(S)$), thus giving us all the vertices of $Vis(S)$.

5 Step 5: Constructing $Vis(S)$

In this step we complete the construction of $Vis(S)$. From Steps 3 and 4 we have all the visible vertices of $Vis(S)$, except those that are horizontal shadows. Moreover, for each visible vertex p we have the rectangles of S that are visible in each face adjacent to p. In order to complete the construction of $Vis(S)$ we must determine all the horizontal shadows in $Vis(S)$, as well as all the adjacency relationships between vertices and edges in $Vis(S)$.

Let B be the set of all visible horizontal dead ends, visible vertical dead ends, visible corner points, and the restricted horizontal shadows of visible corner points. We begin by constructing G, the planar graph determined by the adjacencies of the vertices in B. We construct all the adjacencies between these vertices by performing two calls to a 2-dimensional bucket sorting routine, each time giving B as the set to be sorted. In the

first call we specify that the routine should sort lexicographically by (x, y) coordinates, resulting in the list B_{xy}. In the second call we specify that the routine sort lexicographically by (y, x) coordinates, giving the list B_{yx}. This requires $O(n + k)$ time, where k is the number of vertices in $Vis(S)$. For any vertex p in B we determine the other vertices of B that are adjacent to p in G by examining the immediate predecessors and successors of p in B_{xy} and B_{yx}. In addition, recall that we have for each p the rectangles that are visible in each of the faces of G adjacent to p.

Having so constructed G, we need only extend each restricted horizontal shadow in G to a true horizontal shadow. Let us re-divide the space by the vertical dividing planes used in Step 1, again giving us the slabs Π_1, Π_2, ..., Π_r. From the first part of this Step 5 (explained above) we now have all the vertices in $Vis(S_i \cup E_i)$ as well as all their adjacencies in $Vis(S)$. We say that a vertical segment s in $Vis(S_i \cup E_i)$ is *horizontally exposed from the left* (resp., *right)* if there is a horizontal line that intersects no vertical segments in $Vis(S_i \cup E_i)$ between s and the left (resp., right) boundary of Π_i. For each slab Π_i we define the *left profile* (resp., *right profile*) to be the y-sorted list of vertical segments of $Vis(S_i \cup E_i)$ that are horizontally exposed from the left (resp., right). Let L_i and R_i denote the left and right profiles of Π_i, respectively.

The method for constructing L_i is as follows (the method for R_i is similar). Let $\hat{L}_i$ be the set of all vertices p in $Vis(S_i \cup E_i)$ such that p is adjacent to a horizontal (visible or shadow) segment in $Vis(S_i \cup E_i)$ that intersects the left boundary of Π_i. We can construct $\hat{L}_i$ by examining all the vertices in $Vis(S_i \cup E_i)$ once. Sort the points in $\hat{L}_i$ by decreasing y-coordinates. Each point p in $\hat{L}_i$ determines a segment in L_i, namely, the vertical segment that is adjacent to p. In addition, for each point p we traverse the face of $Vis(S_i \cup E_i)$ that is adjacent to p, but does not contain the vertical segment adjacent to p, to see if it contains a vertical segment horizontally exposed from the left. Note that by the definition of the points in $\hat{L}_i$ we will traverse each such face only once. After we have completed all such traversals, we will have the entire list L_i. This construction takes $O(n + k_i)$ time, where k_i is the number of vertices in $Vis(S_i \cup E_i)$.

We construct a graph that has a node for each L_i and R_i list, and connects each R_i to R_{i-1} and and each L_i to L_{i+1}. Using the terminology of Chazelle and Guibas [5], this graph is a *catalogue graph*. Thus, we can apply the "fractional cascading" technique [5] to build a data structure that consists of augmented lists L'_i and R'_i, for each i in $\{1, 2, ..., r\}$, as well as a number of pointers between consecutive augmented lists, such that given the

position of a point p in some L'_i list this structure allows one to locate p in L_i and L'_{i+1} in $O(1)$ time. The similar property holds for the R'_i lists. Using the method of Chazelle and Guibas [5], this data structure can be constructed in time and space proportional to the total number of elements in all the lists (which is $O(n+k)$).

Let us return to the problem at hand, namely, completing the construction of $Vis(S)$ by finding the true horizontal shadow points of each corner point p that currently has its restricted horizontal shadow falling on a boundary of Π_i, where $p \in \Pi_i$. Let us concentrate on the computation of the true horizontal shadow of a point p whose restricted horizontal shadow falls on the left boundary of Π_i (the method for the case when p's restricted shadow falls on the right boundary of Π_i is similar). We first locate $y(p)$ in R'_{i-1} in $O(\log n)$ time by using the binary search technique. Then, we can locate $y(p)$ in R_{i-1} in $O(1)$ time. If the interval in R_{i-1} in which $y(p)$ falls contains a vertical segment, then we have found the true horizontal shadow of p—simply compute the intersection of the line $y = y(p)$ with this segment. If, on the other hand, this interval is empty of any vertical segments (of $Vis(S_{i-1} \cup E_{i-1})$), then we use the position of $y(p)$ in R'_{i-1} to locate $y(p)$ in R'_{i-2} in $O(1)$ time. We then repeat the above procedure until we locate p's horizontal shadow or run out of lists to search in (in which case p's shadow is $(-\infty, y(p))$). This searching procedure takes at most $O(\log n + r)$ time for each corner point p.

The only thing left, then, is to link each new horizontal shadow into the graph G to give us $Vis(S)$ while removing all the old (restricted) horizontal shadows they replace. To perform this last computation construct a tuple $(x, y, y(p))$, where (x, y) is the upper endpoint of the vertical segment on which the horizontal shadow $(x, y(p))$ of p lies. We can then sort all these tuples in $O(n)$ additional time. Using this sorted list we can complete the construction of $Vis(S)$ in $O(n)$ time. Since there are at most $O(n)$ corner points for which we perform this procedure, the total time for finding these horizontal shadows is $O(n \log n + rn + k)$ time. This completes the algorithm. We summarize the above discussion in the following theorem.

Theorem 5.1: *Given a set S of n isothetic rectangles in $\mathbb{R}^3$, one can solve the hidden-surface elimination problem for S in $O(n \log^2 n + (n^2/r) + rn + k)$ time, where r is any integer parameter and k is the size of the output.* ∎

Corollary 5.2: *One can solve the hidden-surface elimination problem for an isothetic*

collection of rectangles in $O(n^{1.5} + k)$ time.

Proof: Set $r = \sqrt{n}$. ∎

6 Conclusion

In this paper we have given an algorithm for solving the hidden-surface elimination problem for rectangles that runs in $O(n^{1.5} + k)$ time, which is output-sensitive and simultaneously worst-case optimal (for quadratic k). Moreover, our algorithm should be competitive with existing methods for realistic values of n. Of course, solving hidden-surface elimination for rectangles is a special case of the general problem. Can the general hidden-surface elimination problem be solved in time proportional to $k + o(n^2)$?

Acknowledgements

We would like to thank S. Rao Kosaraju and Michael McKenna for helpful discussions.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] B.G. Baumgart, "A Polyhedron Representation for Computer Vision," *Proc. 1975 AFIPS National Computer Conf., 44*, AFIPS Press, 1975, 589–596.

[3] J.L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. on Computers*, Vol. C-29, 1980, 571–577.

[4] M. Bern, "Hidden Surface Removal for Rectangles," *Proc. 4th ACM Symp. on Computational Geometry*, 1988, 183–192.

[5] B. Chazelle and L.J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica*, Vol. 1, No. 2, pp. 133–162.

[6] F. Dévai, "Quadratic Bounds for Hidden-Line Elimination," *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, 269–275.

[7] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, 1987.

[8] H. Edelsbrunner and M.H. Overmars, "Batched Dynamic Solutions to Decomposable Searching Problems," *J. Algorithms*, Vol. 6, 1985, 515–542.

[9] M.T. Goodrich, "A Polygonal Approach to Hidden-Line Elimination," *Proc. of 25th Annual Allerton Conference on Communication, Control, and Computing*, 1987, 849–858.

[10] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science*, 1978, 8–21.

[11] L.J. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Transactions on Graphics*, Vol. 4, 1985, 75–123.

[12] R.H. Güting and T. Ottmann, "New Algorithms For Special Cases of the Hidden Line Elimination Problem," *Computer Vision, Graphics, and Image Processing*, Vol. 40, 1987, 188–204. (A preliminary version of this work appeared in *Proc. Symp. on Theoretical Aspects of Computer Science*, 1985, 161–171.)

[13] M. McKenna, "Worst-case Optimal Hidden-Surface Removal," *ACM Transactions on Graphics*, Vol. 6, No.1, January 1987, 19–28.

[14] D.E. Muller and F.P. Preparata, "Finding the Intersection of Two Convex Polyhedra," *Theoretical Computer Science*, Vol. 7, No. 2, October 1978, 217–236.

[15] O. Nurmi, "A Fast Line-Sweep Algorithm For Hidden Line Elimination," *BIT*, Vol. 25, 1985, 466–472.

[16] T. Ottmann and P. Widmayer, "Solving Visibility Problems by Using Skeleton Structures," *Proc. 11th Symp. on Mathematical Foundations of Computer Science*, 1984, 459–470.

[17] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, NY, 1985.

[18] F.P. Preparata, J.S. Vitter, and M. Yvinec, "Computation of the Axial View of a Set of Isothetic Parallelepipeds," Manuscript, 1987. (Cited in [4].)

[19] A. Schmitt, "On the Time and Space Complexity of Certain Exact Hidden Line Algorithms," Universität Karlsruhe, Fakultät für Informatik, Report 24/81, 1981. (Cited in [12].)

[20] A. Schmitt, "Time and Space Bounds for Hidden Line and Hidden Surface Algorithms," *EUROGRAPHICS '81*, 43–56.

[21] S. Sechrest and D.P. Greenberg, "A Visibility Polygon Reconstruction Algorithm," *ACM Transactions on Graphics*, Vol. 1, No. 1, January 1982, 25–42.

[22] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, Vol. 6, No. 1, March 1974, 1–25.

[23] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.

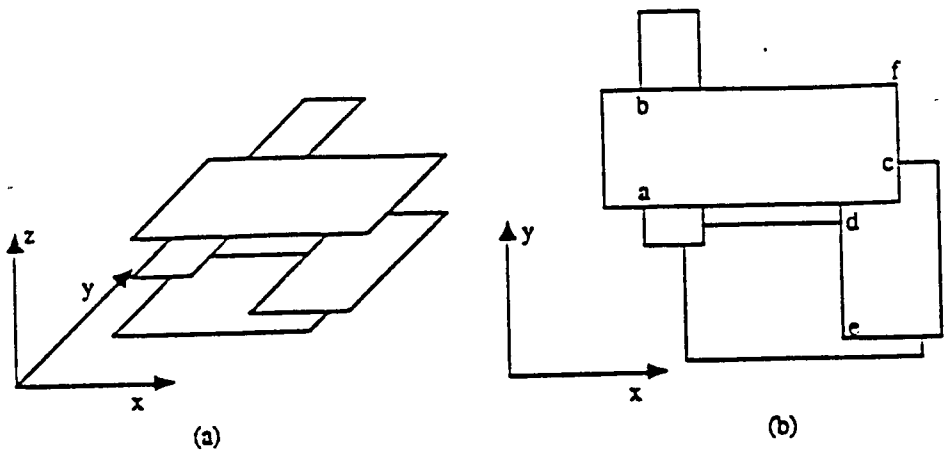

Figure 1

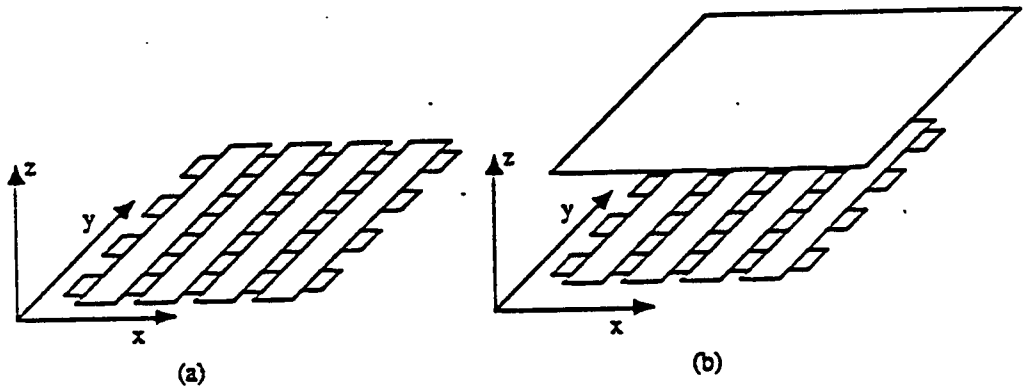

Figure 2

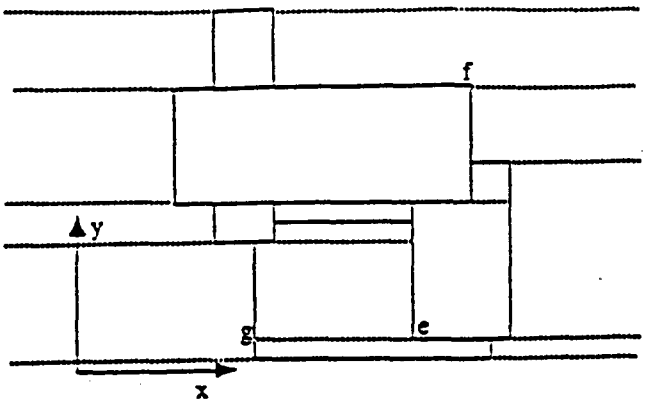

Figure 3

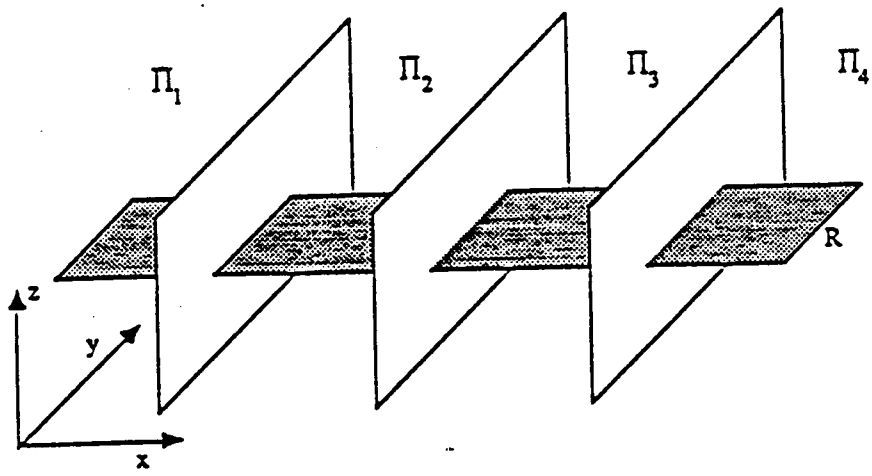

Figure 4

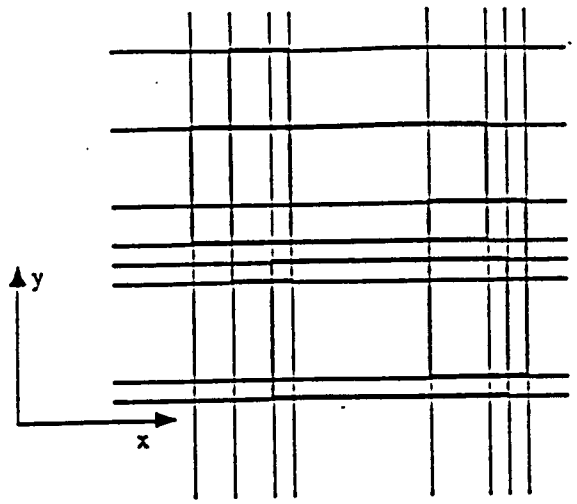

Figure 5

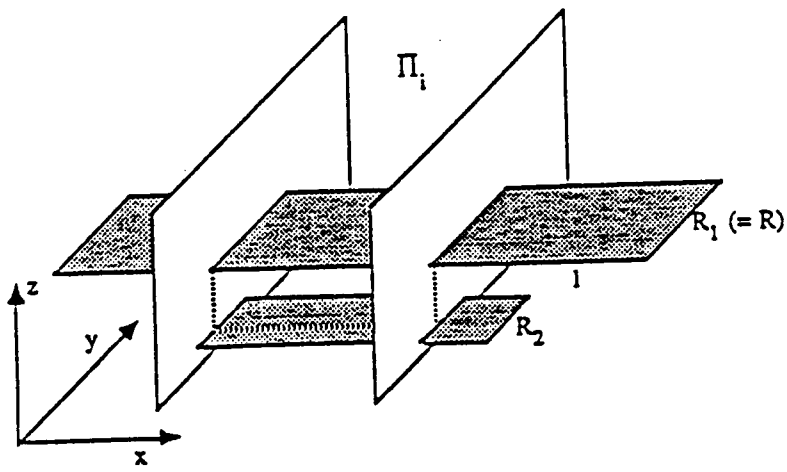

Figure 6